Alire: a Library Repository Manager for the Open Source Ada Ecosystem

Alejandro R. Mosteo
Instituto de Investigación en Ingeniería de Aragón, Mariano Esquillor s/n, 50018, Zaragoza, Spain
Centro Universitario de la Defensa de Zaragoza, 50090, Zaragoza, Spain; email: amosteo@unizar.es

Abstract

Open source movements are main players in today's software landscape. Communities spring around programming languages, providing compilers, tooling and, chiefly, libraries built with these languages. Once a community reaches a certain critical mass, management of available libraries becomes a point of contention. Operating system providers and distributions often support but the most significant or mature libraries so, usually, language communities develop their own cross-platform software management tools. Examples abound with languages such as Python, OCaml, Rust, Haskell and others.

The Ada community has been an exception to date, perhaps due to its smaller open source community. This work presents a working prototype tailored to the Ada compiler available to open source enthusiasts, GNAT. This tool is designed from two main principles: zero-cost infrastructure and a pure Ada work environment. Initially available for Linux-based systems, it relies on the semantic versioning paradigm for dependency resolution and uses Ada specification files to describe project releases and dependencies.

Keywords: Library Management, Dependency resolution, Open Source, Ada 2012.

1 Introduction

"If I have seen further it is by standing on ye sholders of Giants" wrote Sir Isaac Newton in a letter to Robert Hooke [1]. Believers in the virtues of open source licenses may recognize the sentiment; in nowadays rapidly evolving technological landscape, reuse of code is critical to adapt to new technologies, avoid past errors, stay on top of vulnerabilities, and foster collaboration. In the communities built around programming languages this can be seen in the publishing of free software under more or less permissive licenses [2]. Open source programmers want their code to be run and built upon.

However, the availability of code and simplicity of distribution, compared to pre-Internet generalization, has brought with itself its own problems, such as a difficulty to be aware of available libraries, obsolescence of code that becomes unmaintained (a form of *bit rot* [3]) and incompatibilities between versions of a same library, or among different libraries being used simultaneously.

To address those problems, one of the most notable efforts in the open source world are the different Linux distributions. Either based on distribution of source code, like Gentoo [4], or of binaries, like Debian [5], these communities have since long dealt with the problem of packaging consistent systems for different architectures. The difficulty of such a task is captured in the *dependency* or *DLL hell* expressions [6], and one of the most dreaded experiences is ending in a *broken* configuration during an upgrade.

Programmers, however, do not all use the same distribution, nor even the same operating system, since today they can resort to about half a dozen generalist operating systems. Given the polarizing nature of programming languages [7] it is then unsurprising that many languages have seen efforts aimed at providing an easy way of distributing libraries for those languages, as we shall discuss in Section 2. In some cases, like Rust [8], the tool for the distribution of libraries is an integral effort of the team developing the language.

The Ada language, perhaps because of its ties to closed development and today's considered niche place in the language landscape [9], has not seen such a tool appear (to the best of our knowledge), despite the notable amount of open source libraries available [10]. This work presents a tool that could be a first step in this direction, with the main contribution being the tool itself. The tool tries to appeal to the Ada programmer by using native Ada code to describe releases and its dependencies, thus avoiding the need to learn new formats. To use this information, the tool uses self-compilation to incorporate the required data into its catalog of libraries. A contributed byproduct is the semantic versioning library[1] that is used to describe dependencies among releases.

The project started as an informal discussion[2] under the name of Alire (from Ada Library Repository), and this work reflects the view of the author on how a tool that addressed the low-hanging problems of the open source Ada community could be brought to life. The tool itself is termed `alr`,[3] in the vein

[1] `https://github.com/alire-project/semantic_versioning`
[2] `https://github.com/mosteo/alire/issues`
[3] A monospace font is used throughout the paper to denote actual executable commands or logical entities such as files.

of other venerable command-line tools such as `git`, `svn`, etc., and to distinguish it from the general project.

The paper is structured as follows: Section 2 examines the situation in other languages and points the referents taken for this tool. Section 3 introduces the design of `alr` and some use cases. Next, Section 4 presents details about the implementation mechanisms underpinning the design. A brief discussion follows on the open questions this design leaves and, lastly, concluding remarks and future directions close the paper in Section 5.

2 Related Work

The problem of library distribution has been tackled in two main ways, namely distribution of binaries and of source code. The former has the advantage of speed for the user, because it saves the step of compilation. The latter allows the complete tailoring of the building process to one's environment, and reduces the work load and hardware requirements on maintainers. Furthermore, for purely interpreted languages the distribution of sources is unavoidable.

Once libraries are obtained, we see yet two possibilities: installation of packages system-wide, as if they where integral parts of the platform, or local installation in a confined or user sandbox (that sometimes can be the default user environment). In Python's `pip` [11], e. g., libraries are installed globally if run as superuser. If run as a regular user, they will be installed in the user's environment. These two options present to the user a default environment that can become broken [6] when dependencies are improperly managed, and for that reason it is recommended [11] to use a sandbox or virtual environment for each development context (Fig. 1). Some packagers, like OPAM [12] or Nix [13], avoid that duplication by using a common store where individual releases are isolated (i.e., there is not a "current" version of any library).

Mainstream languages such as Java, C, and C++ also have a variety of tools at their disposal, the problem being in this case the lack of a standardized unique (or prevalent) go-to tool. Since these languages do not natively consider the build consistency problem as Ada does, their tools may also include complex building aspects, like Gradle/Maven do for Java [14]. A main player for the C/C++ world, Conan [15], is instead a build-system agnostic package manager that however relies on YAML configuration files and Python scripts, increasing the technical burden.

When one inspects the many solutions out there, like Rust's `cargo`, Python's `pip` and `easy_install`, OCaml's `opam`, D's `dub`, Haskell's stackage and `cabal-install`, to name more examples, a few common traits arise. The backend is usually some kind of database that in its simplest form is merely a set of files under version control in a public repository or in dedicated servers. Submission of new libraries becomes then the merging of a pull request into the stable branch of the catalog. Fetching of a library involves the download of a file bundle or checkout of a particular commit.

The other salient aspect these tools address is dependency resolution. When building a project with a complex set of

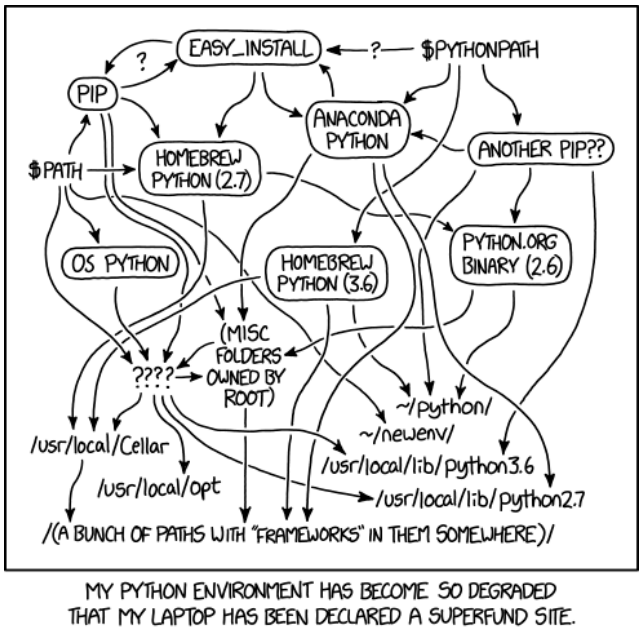

Figure 1: Library management problems have reached the level of Internet running joke (`https://xkcd.com/1987/`)

dependencies, it may happen that two (or more) subprojects depend on the same libraries with some version restrictions. From all the possible combinations, only one that satisfies all dependent projects can be chosen, or if an incompatible request is made a resolution conflict appears. Again, a common approach is to use semantic versions [16] of the form `M.m.p`, where `M` stands for *major* version (one that is backwards incompatible), `m` is the *minor* version (one that is backwards compatible within the same major version) and `p` is a *patch*, a mere bug fix release that should be API compatible with other `M.m` releases. These dependencies are usually represented in some textual description of a release, like key-value lists, JSON, XML, or the own language syntax when it is interpreted.

Semantic versioning is not the only solution to the dependency upgrade issue, but in many cases semantic versions can encompass other paradigms like *calendar versioning* [17] that are less strict in their specification. At a minimum, pinning of versions and careful manual updating is a worst-case scenario that often is unavoidable if projects do not follow a strict backwards-compatible release policy.

3 Design objectives and use cases

For `alr`, after reviewing these solutions, the following decisions were taken, given the constraints of a lack of guaranteed funding and the idiosyncrasies of the Ada language and GNAT build tools:

- The objective is to help develop software, but not to configure the system. Hence, the mode of operation cannot depend on installing the compiled libraries, thus entirely avoiding any possibility of breaking the user's system. The tool operates in user-space and the libraries are stored as source code.

- A sandbox approach is applied for every working project to avoid variations due to build scenarios of a same dependency, which in turn ensures reproducible builds given a compiler/platform combination.
- To not depend on private servers nor live processes, the Alire catalog and code releases are stored in public Version Control System (VCS) services such as GitHub, BitBucket, etc., with which the open source community is used to work.
- New releases are incorporated into Alire by means of a pull request into the catalog repository. Since this is a manual process, at this time Alire can only be considered a curated system.[4]
- An indexed release is described using Ada code that is verified by means of compiling it, relying only on a single specification file that is part of the `alr` source code. The aim is to stay within the Ada realm as much as possible. In its present form, the `alr` tool only requires familiarity with the GNAT [18] compiler.
- Library developers should be minimally impacted for integration into Alire, if at all. This is achieved ultimately by only requiring a GNAT project file (GPR file) that can be created by Alire maintainers without bothering library authors uninterested in this tool.

Ada adopted the idea of library items [19] that can be submitted to the compiler independently. This concept, together with the well-defined dependency and elaboration rules, has spared Ada developers to an extent the quagmire of dependency-building tools such as autoconf, automake and CMake [20]. Given that nowadays there is a single open source Ada compiler, namely GNAT in its GPL and FSF editions, at this time `alr` relies on GNAT aggregate project files to completely manage the building process, without the need to modify the environment. This solution lets programmers use dependencies as usual, merely "with-ing" their project file.

3.1 Components of the Alire project

The Alire project is divided in the following main parts:

- The catalog of projects is a repository hosted under the name of `alire`.[5] It fulfills the same role as, e.g., the `crates.io-index`[6] project in the Rust community. It comprises the database of known projects and the minimal Ada types needed to represent that information. This way, commits to its repository should be for the most part, once development stabilizes, just additions to the catalog.
- The command-line tool available to users to interact with the Alire catalog is named `alr`, as its repository.[7] Again, this allows development on the tool with minimal disturbance to the catalog. It fulfills the role of the `cargo`[8] tool for Rust.
- The indexed code releases from third parties can be in any online repository, with the implicit assumption that the longest lived a repository is, the better. Current free offerings favored by developers are the usual suspects: GitHub, BitBucket, GitLab, etc. Of course, forks of particular releases could be made to ensure high availability.

3.2 Main use cases

Depending on the role of the user, a number of applications can be found for package managers. In its current form `alr` already enables the following use cases:

- Packagers: authors or entities wishing to disseminate their code can publish well-defined releases of their projects with the proper dependencies necessary to build them. Volunteers can also package popular Ada projects to increase their exposure. The Alire catalog knows about all licenses curated by GitHub[9], making explicit the rights granted by publishers.
- Developers: be it with the aim of publishing a project in Alire or not, developers can use `alr` to declare dependencies to be used in their own Ada projects. These dependencies are resolved into a valid solution, their code fetched, and a project file is generated that allows edition/compilation with the GNAT toolchain.
- Final users: despite the 'library' in Alire, more generally any packaged project can be also a binary tool or application. A single `alr get` command allows the retrieval, compilation and verification of target executables of such a binary project.

3.3 Introduction to `alr`

The prototype being discussed in this work is available for testing with a number of representative projects already indexed (see Fig. 6 in last page). Once installed and run without arguments, the user is greeted by the help screen shown in Listing 3.1, which will not be unfamiliar to similar tools users:

```
Ada Library Repository manager (alr)
Usage : alr [global options] command [options] [arguments]

Valid commands:

   build    Upgrade and compile current project
   clean    GPRclean project and dependency cache
   compile  GPRbuild current project
   depend   Manage dependencies of working project
   get      Fetch a project or show its metadata
   init     Create a project or generate its metadata
   list     See indexed projects in database
   pin      Pin dependencies to current versions
   run      Launch a project executable
   search   Search text in project names and properties
   test     Test deployment of releases
   update   Update alire catalog or project dependencies
   version  Shows alr diagnostics
   with     Locate index file of project

Use "alr help <command>" for information about a command
```

Listing 3.1: Help screen of `alr`

[4] The same happens in other languages. For example, in the Haskell community the Stackage project arose as a curated alternative to the `cabal-install` breakage-prone tool.

[5] `https://github.com/alire-project/alire`

[6] `https://github.com/rust-lang/crates.io-index`

[7] `https://github.com/alire-project/alr`

[8] `https://github.com/rust-lang/cargo`

[9] `https://choosealicense.com/`

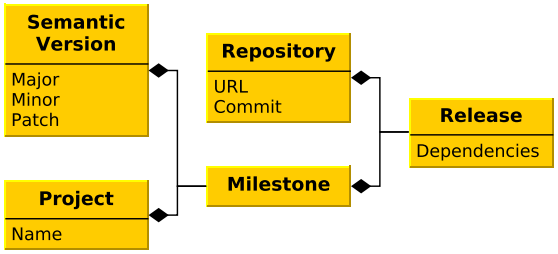

Figure 2: Entities in the Alire catalog.

Before diving into these commands, an explanation on the terminology being used (in the remainder of the paper and in the Alire source code) is in order (see also Fig. 2):

- A *project* refers to what also is typically called a library in the software world; e.g., GtkAda[10], AWS[11], etc.
- A *milestone* is a project name plus a semantic version; i.e., a particular version of a project.
- A *release* is the actual materialization of a milestone, available online and indexed by Alire. A release must provide one or more GPR files that build it.

The most straightforward function of `alr` is to retrieve a particular project and build it. Projects can contain libraries, which are useful to other projects, but also executables, in which case the compilation process will result in one or more executables ready to be run. This is achieved with the `alr get <project>` command. The result will be a folder containing the requested project and its dependencies, so compilation will immediately succeed.

Alternatively, `alr` can create new projects to start easily working within the Alire ecosystem. This is achieved with the `alr init [--bin|--lib] <project>` invocation. Initially the project will not have dependencies; required libraries can be added directly with `alr depend --add <project>` or with especially formatted comments in the user own GPR file.

Any project obtained by each of these two means can be called an alr-enabled or aware project, since it contains a metadata file that allows `alr` to perform its functions. Once within the folder tree of an alr-aware project, we can use the rest of commands (see Fig. 3). The `compile` command launches the `gprbuild` tool with a generated aggregate project file that makes dependencies available without needing to fiddle with paths. The `update` command refreshes the catalog and upgrades the dependencies of the working project.

There are also compound commands that group functions for common combinations: `run` will compile and then launch the resulting executable, whereas `build` will ensure that dependencies are up to date to then compile the project.

The commands interrelations have been designed to guarantee success, in the sense that compilation should always succeed if the requested dependencies are valid. `alr` will also detect the manual addition of new dependencies by the user and fetch them before a new compilation.

[10] `https://github.com/AdaCore/gtkada`
[11] `https://github.com/AdaCore/aws`

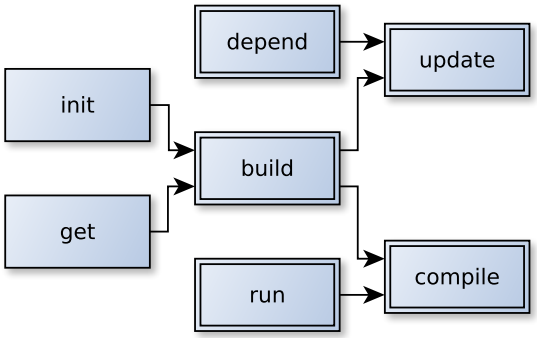

Figure 3: Relationships among commands. Single-frame commands can be used anywhere in the filesystem, whereas double-framed ones are to be used within an alr-enabled project.

To conclude this section, we show how dependencies are represented in a working project. As advanced, this is done in a package specification that can be compiled to verify its correctness, and which is initially generated by `alr`. A dependency on RxAda [21] has been already added.

```
with Alire.Index.Rxada;

package Alr_Deps is

   Current_Root : constant Root := Set_Root (
      "My_Shiny_Project",
      Dependencies =>
         Rxada.Project.Within_Major ("1.1"));

end Alr_Deps;
```

Listing 3.2: Metadata file in an alr-enabled project.

If the user wishes to compile this file directly (instead of through the `alr` tool), it is enough to add the Alire project itself as a dependency of the working project.

Restrictions on dependencies are described using the usual Ada comparison operators, and named functions for the semantic versioning specific operators caret (`'^'`) and tilde (`'~'`). This way there is no possible confusion on what is being asked for (In some implementations, the caret and tilde act differently on pre-1.0 versions). In the example, we request any future version of RxAda that is at least 1.1 but within the same major number, hence backwards-compatible.

4 Implementation details

This section presents some lower level details on `alr` implementation, particularly those aspects that present a specific idiosyncrasy of the tool when compared with its homologues for other languages.

GNAT is currently the only open source Ada compiler available, and its GPR project files are the preferred way to conveniently manage the building process. For these reasons, `alr` takes advantage of these project files, and in particular uses aggregate projects to make available the dependencies to be included in the compilation of a project.

4.1 Alire-mandated files

For `alr` to be able to perform its project-specific commands (see Fig. 3), it needs three critical files to be present:[12]

- `myproject.gpr` (henceforth the project file): this is a regular GPR project file that builds the project. In practice, GNAT projects typically already have one or several project files, so this is not a special requirement. `alr` provides ways of querying the name of these project files for the benefit of client projects.
- `alr_deps.ads` (henceforth the metadata file): this file is used by `alr` as a telltale that it is being run inside an alr-enabled project. It must contain the project name and its dependencies, as already shown in 3.2. It is initially generated by `alr init`, or could be hand-crafted if needed. It can also be regenerated on demand and manipulated through `alr depend`.
- `alr_build.gpr` (henceforth the environment file): this file is generated by `alr` to set up the environment paths required to find any projects the current project depends on. It can also be used to work in the GNAT GPS IDE.

Of these three files, the only one that is entirely the responsibility of the project author (or maintainer) is the `myproject.gpr` one. Its contents are arbitrary, as long as they succeed in building the library or executable. At a minimum, they must point the compiler to the source files of the project. On the other extreme, `alr_build.gpr` is regenerated by `alr` whenever necessary to properly configure the building environment (namely, whenever dependencies change or the file is not found). `alr_deps.ads` lies in the middle, since it is initially generated by `alr` but it must by tailored by the developer to their needs to indicate their dependencies.

Finally note that, for the inclusion of a project into the Alire catalog, only the project file is needed, since the contents of the metadata file will appear in the Alire index itself (see Listing 4.1), and the environment file is regenerated from that information. Each Alire index file contains the releases for the project named as the enclosing package. Besides textual versions, dependencies within the index can be specified using other indexed releases:

```
with Alire.Index.Libhello;

package Alire.Index.Hello is

   function Project is new Catalogued_Project
     ("""Hello, world!"" demonstration project");

   Repo : constant URL :=
      "https://github.com/alire-project/hello.git";

   V_1_0_1 : constant Release :=
      Project.Register
        (V ("1.0.1"),
         Git (Repo, "8cac0afdd"),
         Dependencies => Libhello.V_1_0.Within_Major);
   -- V_1_0 is an existing release of Libhello

end Alire.Index.Hello;
```

Listing 4.1: Release in the index with one dependency.

[12] "myproject" is a placeholder for an actual project name.

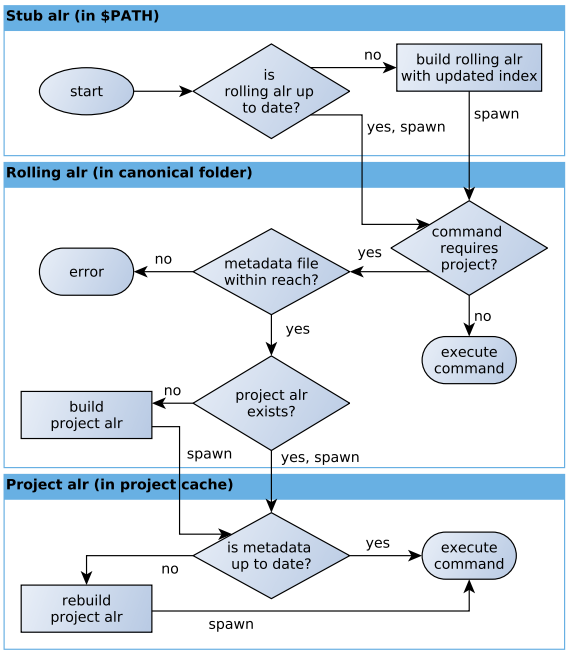

Figure 4: Launch sequence of `alr` for command execution.

A rich set of operations exists that allows the expression of not only simple dependencies, but also of conditional dependencies on the compiler version, platform properties, availability of native packages, and so forth. Also, to simplify indexing and clarifying the declarations, a base release can be taken as a template and modified with "extending" and "replacing" operations. For details the reader is directed to examples in the Alire database itself.[13]

4.2 Self-compilation of `alr` and working projects

Package managers are expected to have an up-to-date catalog, and also that the tool itself is up-to-date. In this case, a catalog update could be achieved in several ways: parsing text files that contain some specific format, or loading a binary database, for example. However, maintaining the tool up-to-date will involve compiling it from updated sources and replace the current executable. Also, incorporating the dependencies of a working project (parsing the `alr_deps.ads` file) would need either a custom parser or compilation and processing with ASIS [22] or a similar technology like libadalang [23].

As an alternative, `alr` solves all these necessities in a single and perhaps uncommon way: whenever the need is detected, `alr` recompiles itself, incorporating into the build fresh metadata and updated index files. This way all needed and up-to-date information is incorporated into `alr` without the need to parse any external files, since the compiler already does the work for us.

To manage this process of self-compilation, up to three different `alr` executables may exist and be called in succession, with specific responsibilities. All three come from the same sources, with a small set of variations for the specific purposes, and are deployed in different locations (see Fig. 4):

[13] Syntax examples: `https://github.com/alire-project/alire/blob/master/index/alire-index-alire.ads`

1. A *stub* `alr` is built during installation (or could be provided by the platform). This binary is never recompiled, acting as a fallback, and will typically be in the system PATH. Its purpose is to build the *rolling* `alr` from updated sources (for example to include new index releases) and launch it.
2. The *rolling* `alr` has an updated index and can already execute commands that are not project-specific (see Fig. 3). If, however, the command requires a project, and furthermore a project metadata file is in scope, then it builds and launchs a *project* `alr`, incorporating into the build the metadata file of the project (and so compiling-in the project dependencies).
3. The *project* `alr` contains the project metadata and is able to carry out project-specific actions. Prior to doing so, it checks its self-consistency by comparing the hash of the metadata file in scope with a hash stored internally that was computed by the *rolling* `alr`. If they do not match, this means that the *project* `alr` is outdated, in which case it is rebuilt with current metadata and launched to take over the command.

4.3 Final example

The creation of new projects from templates or downloading of releases do not really merit any special discussion, since they do not pose particular technological challenges. However, inspecting the filesystem after the issuing of an `alr get --compile hello` command will allow to bring into focus everything that has been reported up to this point. This command simultaneously fetches a project and its dependencies, generates the needed files and builds the whole configuration. The project itself is a plain "Hello, world!" example artificially split into having to depend on a library (libhello) that performs the actual output to the terminal.

Fig. 5 shows the relevant parts of a filesystem in which such a command were issued in the user's home folder. From top to bottom, the following relevant folders and files can be located:

- The stub alr can be anywhere in the user's path, here shown in `/usr/bin/alr`.
- `$XDG_CONFIG_HOME/alire/` is the canonical location in which updated sources are checked out. Inside, the `alire/index/` folder contains the catalog files, and the most recently built rolling `alr` executable is found in `alr/bin/alr`.
- `hello_1.0.1_65725c20/` is the folder in which the requested project, `hello`, has been deployed. The semantic version and abbreviated commit hash are appended to univocally identify the project. The project own organization is an internal affair of the project author; in this example the minimal project and main files are shown. Alire files can be found inside the `alire` subfolder:
- `<project>/alire/` contains firstly the metadata and environment files. The metadata file can be manually edited or manipulated through `alr depend`. The build file is regenerated on changes to the metadata file, and is useful to launch builds, or to edit from GPS.

```
/
├── usr/bin/alr (stub)
└── $HOME/
    ├── $XDG_CONFIG_HOME/
    │   └── alire/
    │       ├── alire/index/ (contents omitted)
    │       └── alr/
    │           ├── bin/alr (rolling)
    │           └── src/ (contents omitted)
    └── hello_1.0.1_65725c20/
        ├── alire
        │   ├── alr_build.gpr
        │   ├── alr_deps.ads
        │   └── cache/
        │       ├── projects/
        │       │   └── libhello_1.0.0_ce78e770/ (contents omitted)
        │       └── session/
        │           ├── alr (project)
        │           ├── alr_deps.ads
        │           └── alr-session.ads
        ├── hello.adb
        └── hello.gpr
```

Figure 5: Filesystem details (comments parenthesized).

- `<project>/alire/cache/`, finally, contains files that the user does not need to directly know about, and that can be deleted at any time since `alr` can download or generate them again as needed. The `projects/` folder contains downloaded dependencies (in this case a particular release of the libhello dependency). The `session/` folder contains generated files for the project build of `alr`. This folder is passed as-is to GPRbuild so it finds the following files:
  - `alr_deps.ads` is a copy of the metadata file.
  - `alr-session.ads` is a file generated at every rebuild that stores the hash of the current metadata, among other information.
  - `alr` is the built project `alr`.

The generated environment file for this example is shown in Listing 4.2:

```
aggregate project Alr_Build is

   for Project_Files use (" ../ hello.gpr");
   -- Root project being compiled

   for Project_Path use
     ("cache/projects/libhello_1.0.0_ce78e770");
   -- Project file paths of dependencies

   for External ("ALIRE") use "True";
   -- Flag that this is an Alire build

end Alr_Build;
```

Listing 4.2: The environment file is a GPR aggregate project file.

4.4 Discussion

At the time of this writing `alr` offers commands and features that make feasible the distribution and reuse of Ada libraries exclusively using Ada tooling and free, public repositories. (Appropriate index files could also enable its use within private environments.) Rich dependencies can be expressed conditionally, and native packages can be used where available. Finally, triggers allow the execution of external programs

at the post-fetch and post-build stages. These features are enough to cover a wide range of needs expected from typical source-oriented package managers.

Substantive effort has been devoted to the testing of both the tool and the packaged projects: through continuous integration, every `alr` master commit is tested to vet proper operation of the `alr` commands, and to verify that releases build properly in supported platforms (which include Debian testing, Ubuntu LTS, and GPL 2017 at this time). Outstanding open issues are:

Windows port: although technically not difficult, the lack of a platform package manager would limit the initial availability of projects with complex dependencies (e.g., GtkAda) that are natively supported in Linux variants.

Cross-platform builds: given the relevance of Ada in the embedded world, this is a feature that has already been pointed out to be important, and that is slated for inclusion in a future release if ongoing interest in `alr` is evidenced.

Given the presented design, compilation times of `alr` itself could be a point of contention since such compilations happen every time the metadata file changes (i.e., whenever dependencies are added or removed). To assess that point, experimental runs were conducted for different catalog sizes. However, since only a few files are recompiled every time (session and metadata files, and one body that uses them in `alr`), the impact is mostly limited to the time it takes to redo the binding and linking. Times measured with a middle-range[14] computer are shown in Table 1. Although not negligible, there is wiggle room until the issue becomes a pressing bottleneck.

	Indexed files		
Releases per file	100	1000	10000
1	1.82	3.73	34.09
10	1.94	4.52	44.83

Table 1: Average times (in seconds) for 100 `alr` recompilations after metadata changes, for different number of files in the catalog and releases per file. Compiler version was GNAT GPL 2017 using `-j0` switch.

5 Conclusions

This work presented an Ada tool, its underlying design, and supporting infrastructure that facilitates easy dissemination and reuse of third-party Ada projects. This is achieved by indexing and tagging code releases in public repositories with a semantic version, which in turn enables the possibility of dependency resolution and easy upgrades. The whole setup only requires a recent GNAT Ada compiler and enables effortless downloading and compilation of indexed projects.

The design is based around a metadata file which is itself written in Ada and incorporated into the tool by recompilation triggered by the tool itself, when needed. This process allows users and developers of the tool alike to remain within the realm of pure Ada code. The Ada syntax employed in index files has a rich feature set that allows the expression of complex conditional dependencies on the availability of native packages or other platform characteristics. This syntax is however only relevant to packagers, since users can add or remove dependencies through tool commands.

Alire is available under an open source license to interested parties at `https://github.com/alire-project`.

[14] Intel® Core™ i3-2015 (4 execution threads), 16GB RAM, SSHD disk.

Acknowledgments

This work has been supported by projects ROBOCHALLENGE (DPI2016-76676-R-AEI/FEDER-UE), ESTER (CUD2017-18), SIVINDRA (UZCUD2017-TEC-06) and ROPERT (DGA-T45_17R/FSE). The author thanks the regulars at `comp.lang.ada` for insightful discussions on the topic.

References

[1] I. Newton, H. W. Turnbull, and J. F. Scott (1959), *The correspondence of Isaac Newton / edited by H.W. Turnbull*. Published for the Royal Society at the University Press Cambridge.

[2] C. Peterson, *How I coined the term 'open source'*. Available at `https://opensource.com/article/18/2/coining-term-open-source-software`.

[3] M. Odersky and A. Moors (2009), *Fighting bit rot with types (experience report: Scala collections*, in LIPIcs-Leibniz Int. Proceedings in Informatics, vol. 4, Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[4] G. K. Thiruvathukal (2004), *Gentoo linux: the next generation of linux*, Computing in science & engineering, vol. 6, no. 5, pp. 66–74.

[5] L. Brenta and S. Leake, *Debian policy for Ada*. Available at `https://people.debian.org/~lbrenta/debian-ada-policy.html`.

[6] S. Eisenbach, V. Jurisic, and C. Sadler (2003), *Managing the evolution of .NET programs*, in International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 185–198, Springer.

[7] A. Stefik and S. Hanenberg (2014), *The programming language wars: Questions and responsibilities for the programming language community*, in Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, pp. 283–299, ACM.

[8] N. D. Matsakis and F. S. Klock II (2014), *The Rust language*, ACM SIGAda Ada Letters, vol. 34, no. 3, pp. 103–104.

[9] D. Hamilton and P. Pape (2017), *20 years after the mandate*, CrossTalk, p. 15.

[10] Ada Information Clearinghouse, *Ada free tools and libraries*. Available at `http://www.adaic.org/ada-resources/tools-libraries/`.

```
$ alr search --list
NAME                                VERSION             DESCRIPTION
ada_lua                             0.0.0-5.3           An Ada binding for Lua
adacurses                           6.0.0               Wrapper on different packagings of NcursesAda
adayaml                             0.3.0               Experimental YAML 1.3 implementation in Ada
adayaml.server                      0.3.0               Server component
alire                               0.4.0               Alire project catalog and support files
alr                                 0.4.0               Command-line tool from the Alire project
apq                                 3.2.0               APQ Ada95 Database Library (core)
aunit                               2017.0.0            Ada unit test framework
eagle_lander                        1.0.0               Apollo 11 lunar lander simulator (Ada/Gtk/Cairo)
globe_3d                            20180111.0.0        GL Object Based Engine for 3D in Ada
hangman                             1.0.0               Hangman game for the console
hello                               1.0.1               "Hello, world!" demonstration project
libadacrypt                         0.8.7               A crypto library for Ada with a nice API
libhello                            1.0.0               "Hello, world!" demonstration project support library
mathpaqs                            20180114.0.0        A collection of mathematical, 100% portable, packages
openglada                           0.6.0               Thick Ada binding for OpenGL and GLFW
pragmarc                            2017.2007.0         PragmAda Reusable Components (PragmARCs)
rxada                               0.1.0               RxAda port of the Rx framework
sdlada                              2.3.1               Ada 2012 bindings to SDL 2
semantic_versioning                 0.3.1               Semantic Versioning in Ada
simple_components.connections       4.27.0              Simple Components (clients/servers)
simple_components.connections.ntp   4.27.0              Simple Components (Network Time Protocol)
simple_components.connections.secure 4.27.0             Simple Components (clients/servers over TLS)
simple_components.core              4.27.0              Simple Components (core components)
simple_components.odbc              4.27.0              Simple Components (ODBC bindings)
simple_components.sqlite            4.27.0              Simple Components (SQLite)
simple_components.strings_edit      4.27.0              Simple Components (strings)
simple_components.tables            4.27.0              Simple Components (tables)
simple_logging                      1.0.0               Simple logging to console
steamsky                            2.1.0-dev           Roguelike in sky with steampunk theme
whitakers_words                     2017.9.10           William Whitaker's WORDS, a Latin dictionary
xml_ez_out                          1.6.0               Creation of XML-formatted output from Ada programs
```

Figure 6: Current alr catalog as listed by the `alr search` command.

[11] K. Reitz and T. Schlusser (2016), *The Hitchhiker's Guide to Python: Best Practices for Development*, O'Reilly Media, Inc.

[12] F. Tuong, F. Le Fessant, and T. Gazagnaire (2012), *OPAM: an OCaml package manager*, in SIGPLAN OCaml Users and Developers Workshop.

[13] E. Dolstra and A. Löh (2008), *NixOS: A purely functional linux distribution*, ACM Sigplan Notices, vol. 43, no. 9, pp. 367–378.

[14] B. Muschko (2014), *Gradle in action*, Manning.

[15] Conan, the C / C++ package manager for developers. Available at `https://conan.io/`.

[16] S. Raemaekers, A. Van Deursen, and J. Visser (2014), *Semantic versioning versus breaking changes: A study of the maven repository*, in 14th Int. Conf. on Source Code Analysis and Manipulation (SCAM), pp. 215–224.

[17] M. Hashemi (2016), *Calendar versioning*. Available at `http://calver.org/`.

[18] E. Schonberg and B. Banner (1994), *The GNAT project: a GNU-Ada 9X compiler*, in Proceedings of the conference on TRI-Ada'94, pp. 48–57, ACM.

[19] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, P. Leroy, and E. Schonberg (2014), *Ada 2012 Reference Manual. Language and Standard Libraries: Int. Standard ISO/IEC 8652/2012 (E)*, vol. 8339, Springer.

[20] J. Al-Kofahi, T. N. Nguyen, and C. Kästner (2016), *Escaping AutoHell: a vision for automated analysis and migration of autotools build systems*, in 4th Int. Workshop on Release Engineering, pp. 12–15, ACM.

[21] A. R. Mosteo (2017), *Rxada: An Ada implementation of the ReactiveX API*, in Ada-Europe International Conference on Reliable Software Technologies, pp. 153–166, Springer.

[22] C. Colket (1995), *Ada semantic interface specification ASIS*, ACM SIGAda Ada Letters, no. 4, pp. 50–63.

[23] P.-M. de Rodat and R. Amiard (2018), *Easy ada tooling with libadalang*, in Ada-Europe International Conference on Reliable Software Technologies.